Code

The tutorial code for Part 2 lives [here](https://github.com/wendykan/DeepLearningMovies/blob/master/Word2Vec_AverageVectors.py).

Introducing Distributed Word Vectors

This part of the tutorial will focus on using distributed word vectors created by the Word2Vec algorithm. (For an overview of deep learning, as well as pointers to some additional tutorials, see the "What is Deep Learning?" page).

Parts 2 and 3 assume more familiarity with Python than Part 1. We developed the following code on a dual-core Macbook Pro, however, we have not yet run the code successfully on Windows. If you are a Windows user and you get it working, please leave a note on how you did it in the forum! For more detail, see the "Setting Up Your System" page.

[Word2vec](https://code.google.com/p/word2vec/), published by Google in 2013, is a neural network implementation that learns [distributed representations](http://www.cs.toronto.edu/~bonner/courses/2014s/csc321/lectures/lec5.pdf) for words. Other deep or recurrent neural network architectures had been proposed for learning word representations prior to this, but the major problem with these was the long time required to train the models. Word2vec learns quickly relative to other models.

Word2Vec does not need labels in order to create meaningful representations. This is useful, since most data in the real world is unlabeled. If the network is given enough training data (tens of billions of words), it produces word vectors with intriguing characteristics. Words with similar meanings appear in clusters, and clusters are spaced such that some word relationships, such as analogies, can be reproduced using vector math. The famous example is that, with highly trained word vectors, "king - man + woman = queen."

Check out [Google's code, writeup, and the accompanying papers](https://code.google.com/p/word2vec/). [This presentation](https://docs.google.com/file/d/0B7XkCwpI5KDYRWRnd1RzWXQ2TWc/edit) is also helpful. The original code is in C, but it has since been ported to other languages, including Python. We encourage you to play with the original C tool, but be warned that it is not user-friendly if you are a beginning programmer (we had to manually edit the header files to compile it).

Recent work out of Stanford has also [applied deep learning to sentiment analysis](http://nlp.stanford.edu/sentiment/); their code is available in Java. However, their approach, which relies on sentence parsing, cannot be applied in a straightforward way to paragraphs of arbitrary length.

Distributed word vectors are powerful and can be used for many applications, particularly word prediction and translation. Here, we will try to apply them to sentiment analysis.

Using word2vec in Python

In Python, we will use the excellent implementation of word2vec from the gensim package. If you don't already have gensim installed, you'll need to [install it](http://radimrehurek.com/gensim/install.html). There is an excellent tutorial that accompanies the Python Word2Vec implementation, [here](http://radimrehurek.com/2014/02/word2vec-tutorial/).

Although Word2Vec does not require graphics processing units (GPUs) like many deep learning algorithms, it is compute intensive. Both Google's version and the Python version rely on multi-threading (running multiple processes in parallel on your computer to save time). ln order to train your model in a reasonable amount of time, you will need to install cython ([instructions here](http://docs.cython.org/src/quickstart/install.html)). Word2Vec will run without cython installed, but it will take days to run instead of minutes.

Preparing to Train a Model

Now down to the nitty-gritty! First, we read in the data with pandas, as we did in Part 1. Unlike Part 1, we now use unlabeledTrain.tsv, which contains 50,000 additional reviews with no labels. When we built the Bag of Words model in Part 1, extra unlabeled training reviews were not useful. However, since Word2Vec can learn from unlabeled data, these extra 50,000 reviews can now be used.

import pandas as pd  
  
# Read data from files   
train = pd.read\_csv( "labeledTrainData.tsv", header=0,   
 delimiter="\t", quoting=3 )  
test = pd.read\_csv( "testData.tsv", header=0, delimiter="\t", quoting=3 )  
unlabeled\_train = pd.read\_csv( "unlabeledTrainData.tsv", header=0,   
 delimiter="\t", quoting=3 )

# Verify the number of reviews that were read (100,000 in total)  
print "Read %d labeled train reviews, %d labeled test reviews, " \  
 "and %d unlabeled reviews\n" % (train["review"].size,   
 test["review"].size, unlabeled\_train["review"].size )

The functions we write to clean the data are also similar to Part 1, although now there are a couple of differences. First, to train Word2Vec it is better not to remove stop words because the algorithm relies on the broader context of the sentence in order to produce high-quality word vectors. For this reason, we will make stop word removal optional in the functions below. It also might be better not to remove numbers, but we leave that as an exercise for the reader.

# Import various modules for string cleaning  
from bs4 import BeautifulSoup  
import re  
from nltk.corpus import stopwords  
  
def review\_to\_wordlist( review, remove\_stopwords=False ):  
 # Function to convert a document to a sequence of words,  
 # optionally removing stop words. Returns a list of words.  
 #  
 # 1. Remove HTML  
 review\_text = BeautifulSoup(review).get\_text()  
 #   
 # 2. Remove non-letters  
 review\_text = re.sub("[^a-zA-Z]"," ", review\_text)  
 #  
 # 3. Convert words to lower case and split them  
 words = review\_text.lower().split()  
 #  
 # 4. Optionally remove stop words (false by default)  
 if remove\_stopwords:  
 stops = set(stopwords.words("english"))  
 words = [w for w in words if not w in stops]  
 #  
 # 5. Return a list of words  
 return(words)

Next, we want a specific input format. Word2Vec expects single sentences, each one as a list of words. In other words, the input format is a list of lists.

It is not at all straightforward how to split a paragraph into sentences. There are all kinds of gotchas in natural language. English sentences can end with "?", "!", """, or ".", among other things, and spacing and capitalization are not reliable guides either. For this reason, we'll use NLTK's punkt tokenizer for sentence splitting. In order to use this, you will need to install NLTK and use nltk.download() to download the relevant training file for punkt.

# Download the punkt tokenizer for sentence splitting  
import nltk.data  
nltk.download()   
  
# Load the punkt tokenizer  
tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')  
  
# Define a function to split a review into parsed sentences  
def review\_to\_sentences( review, tokenizer, remove\_stopwords=False ):  
 # Function to split a review into parsed sentences. Returns a   
 # list of sentences, where each sentence is a list of words  
 #  
 # 1. Use the NLTK tokenizer to split the paragraph into sentences  
 raw\_sentences = tokenizer.tokenize(review.strip())  
 #  
 # 2. Loop over each sentence  
 sentences = []  
 for raw\_sentence in raw\_sentences:  
 # If a sentence is empty, skip it  
 if len(raw\_sentence) > 0:  
 # Otherwise, call review\_to\_wordlist to get a list of words  
 sentences.append( review\_to\_wordlist( raw\_sentence, \  
 remove\_stopwords ))  
 #  
 # Return the list of sentences (each sentence is a list of words,  
 # so this returns a list of lists  
 return sentences

Now we can apply this function to prepare our data for input to Word2Vec (this will take a couple minutes):

sentences = [] # Initialize an empty list of sentences

print "Parsing sentences from training set"

for review in train["review"]:

sentences += review\_to\_sentences(review, tokenizer)

print "Parsing sentences from unlabeled set"

for review in unlabeled\_train["review"]:

sentences += review\_to\_sentences(review, tokenizer)

You may get a few warnings from BeautifulSoup about URLs in the sentences. These are nothing to worry about (although you may want to consider removing URLs when cleaning the text).

We can take a look at the output to see how this differs from Part 1:

>>> # Check how many sentences we have in total - should be around 850,000+  
... print len(sentences)  
857234  
  
>>> print sentences[0]  
[u'with', u'all', u'this', u'stuff', u'going', u'down', u'at', u'the', u'moment', u'with', u'mj', u'i', u've', u'started', u'listening', u'to', u'his', u'music', u'watching', u'the', u'odd', u'documentary', u'here', u'and', u'there', u'watched', u'the', u'wiz', u'and', u'watched', u'moonwalker', u'again']  
  
>>> print sentences[1]  
[u'maybe', u'i', u'just', u'want', u'to', u'get', u'a', u'certain', u'insight', u'into', u'this', u'guy', u'who', u'i', u'thought', u'was', u'really', u'cool', u'in', u'the', u'eighties', u'just', u'to', u'maybe', u'make', u'up', u'my', u'mind', u'whether', u'he', u'is', u'guilty', u'or', u'innocent']

A minor detail to note is the difference between the "+=" and "append" when it comes to Python lists. In many applications the two are interchangeable, but here they are not. If you are appending a list of lists to another list of lists, "append" will only append the first list; you need to use "+=" in order to join all of the lists at once.

Training and Saving Your Model

With the list of nicely parsed sentences, we're ready to train the model. There are a number of parameter choices that affect the run time and the quality of the final model that is produced. For details on the algorithms below, see the word2vec [API documentation](http://radimrehurek.com/gensim/models/word2vec.html) as well as the [Google documentation](https://code.google.com/p/word2vec/).

* Architecture: Architecture options are skip-gram (default) or continuous bag of words. We found that skip-gram was very slightly slower but produced better results.
* Training algorithm: Hierarchical softmax (default) or negative sampling. For us, the default worked well.
* Downsampling of frequent words: The Google documentation recommends values between .00001 and .001. For us, values closer 0.001 seemed to improve the accuracy of the final model.
* Word vector dimensionality: More features result in longer runtimes, and often, but not always, result in better models. Reasonable values can be in the tens to hundreds; we used 300.
* Context / window size: How many words of context should the training algorithm take into account? 10 seems to work well for hierarchical softmax (more is better, up to a point).
* Worker threads: Number of parallel processes to run. This is computer-specific, but between 4 and 6 should work on most systems.
* Minimum word count: This helps limit the size of the vocabulary to meaningful words. Any word that does not occur at least this many times across all documents is ignored. Reasonable values could be between 10 and 100. In this case, since each movie occurs 30 times, we set the minimum word count to 40, to avoid attaching too much importance to individual movie titles. This resulted in an overall vocabulary size of around 15,000 words. Higher values also help limit run time.

Choosing parameters is not easy, but once we have chosen our parameters, creating a Word2Vec model is straightforward:

# Import the built-in logging module and configure it so that Word2Vec   
# creates nice output messages  
import logging

logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',\  
 level=logging.INFO)

# Set values for various parameters

num\_features = 300 # Word vector dimensionality

min\_word\_count = 40 # Minimum word count

num\_workers = 4 # Number of threads to run in parallel

context = 10 # Context window size

downsampling = 1e-3 # Downsample setting for frequent words

# Initialize and train the model (this will take some time)  
from gensim.models import word2vec

print "Training model..."

model = word2vec.Word2Vec(sentences, workers=num\_workers, \  
 size=num\_features, min\_count = min\_word\_count, \

window = context, sample = downsampling)

# If you don't plan to train the model any further, calling   
# init\_sims will make the model much more memory-efficient.

model.init\_sims(replace=True)

# It can be helpful to create a meaningful model name and   
# save the model for later use. You can load it later using Word2Vec.load()

model\_name = "300features\_40minwords\_10context"

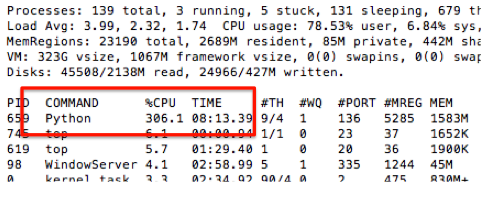
model.save(model\_name)

On a dual-core Macbook Pro, this took less than 15 minutes to run using 4 worker threads. However, it will vary depending on your computer. Fortunately, the logging functionality prints informative messages.

If you are on a Mac or Linux system, you can use the "top" command from within Terminal (not from within Python) to see if your system is successfully parallelizing while the model is training. Type

> top -o cpu

into a terminal window while the model is training. With 4 workers, the first process in the list should be Python, and it should show 300-400% CPU usage.



If your CPU usage is lower, it may be that cython is not working correctly on your machine.

Exploring the Model Results

Congratulations on making it successfully through everything so far! Let's take a look at the model we created out of our 75,000 training reviews.

The "doesnt\_match" function will try to deduce which word in a set is most dissimilar from the others:

>>> model.doesnt\_match("man woman child kitchen".split())  
'kitchen'

Our model is capable of distinguishing differences in meaning! It knows that men, women and children are more similar to each other than they are to kitchens. More exploration shows that the model is sensitive to more subtle differences in meaning, such as differences between countries and cities:

>>> model.doesnt\_match("france england germany berlin".split())  
'berlin'

... although with the relatively small training set we used, it's certainly not perfect:

>>> model.doesnt\_match("paris berlin london austria".split())  
'paris'

We can also use the "most\_similar" function to get insight into the model's word clusters:

>>> model.most\_similar("man")  
[(u'woman', 0.6056041121482849), (u'guy', 0.4935004413127899), (u'boy', 0.48933547735214233), (u'men', 0.4632953703403473), (u'person', 0.45742249488830566), (u'lady', 0.4487500488758087), (u'himself', 0.4288588762283325), (u'girl', 0.4166809320449829), (u'his', 0.3853422999382019), (u'he', 0.38293731212615967)]  
  
>>> model.most\_similar("queen")  
[(u'princess', 0.519856333732605), (u'latifah', 0.47644317150115967), (u'prince', 0.45914226770401), (u'king', 0.4466976821422577), (u'elizabeth', 0.4134873151779175), (u'antoinette', 0.41033703088760376), (u'marie', 0.4061327874660492), (u'stepmother', 0.4040161967277527), (u'belle', 0.38827288150787354), (u'lovely', 0.38668593764305115)]

Given our particular training set, it's not surprising that "Latifah" is a top hit for similarity with "Queen".

Or, more relevant for sentiment analysis:

>>> model.most\_similar("awful")  
[(u'terrible', 0.6812670230865479), (u'horrible', 0.62867271900177), (u'dreadful', 0.5879652500152588), (u'laughable', 0.5469599962234497), (u'horrendous', 0.5167273283004761), (u'atrocious', 0.5115568041801453), (u'ridiculous', 0.5104714632034302), (u'abysmal', 0.5015234351158142), (u'pathetic', 0.4880446791648865), (u'embarrassing', 0.48272213339805603)]

So it seems we have a reasonably good model for semantic meaning - at least as good as Bag of Words. But how can we use these fancy distributed word vectors for supervised learning? The next section takes a stab at that.